下载APP

# 22 | 增强编译器前端功能第1步：再识数据流分析技术

2021-09-27 宫文学

《手把手带你写一门编程语言》 课程介绍 >

**讲述：宫文学**

时长 18:16 大小 16.74M

你好，我是宫文学。

到目前为止，实现一门计算机语言的流程，我已经带你完整地走了一遍：从编译器前端的技术，到 AST 解释器，再到字节码虚拟机，最后生成了汇编代码并编译成了可执行文件。在这个过程中，我们领略了沿途的风光，初步了解了实现一门计算机语言的各种关键技术。

可是，我们在第一部分起步篇里，都是顾着奋力攀爬，去开出一条路来。可是这条路实在有点窄，是条羊肠小道，现在我们就需要把它拓宽一下。也就是把我们 PlayScript 语言特性增强一下，拓宽我们的知识面。

这个拓宽的方式呢，我选择的是围绕**数据类型**这条主线进行。这是因为，现代计算机语言的很多特性，都是借助类型体系来呈现的。

不知道你注意到没有，到目前为止，除了最早的 AST 解释器以外，我们的后几个运行机制都只支持整型。所以，我们要在第二部分进阶篇中让 PlayScript 支持浮点型、字符串和自定义对象等数据类型。做这些工作的目的，不仅仅是增加我们语言支持的数据类型，而且，随着你对字符串类型、和自定义对象等类型的学习，你也会对对象的处理能力，包括对象的属性、方法、对象内存的自动管理等知识有更深刻的理解。

为了降低工作量，我后面的课程主要实现的是实现静态编译的版本。因为这种运行机制涉及的知识点比较广，并且我的目标之一就是要**实现一个高效的、静态编译的 TypeScript 版本，**到这里我的目标也算达到了。如果你有兴趣，你也可以把字节码虚拟机版本扩展一下，用于支持对象特性等高级特性。

不过，在我们开启第二段征程之时，我们需要回到编译器的前端部分，把词法分析、语法分析和语义分析等功能都增强一下，以便支持后面的语法特性。在这个过程中，你会学习到前端的数据流分析技术、前端的优化技术和类型计算方面的知识点，让你的编译器前端的知识得到迭代提升。

那么，首先我们就来看看，我们语言在编译器前端功能方面的现状，找找增强这些功能的办法。

## 编译器前端功能上的现状

编译器的前端，也就是词法分析、语法分析和语义分析功能。我们逐个回顾一下，看看我们现在做到怎么样了。

首先来看我们的**词法分析**功能，跟语法分析和语义分析功能相比，它应该算是最完善的了。为什么这么说呢？因为我们目前的语言特性已经涉及到了大部分种类的 Token，这些词法分析器都能提供。但相对来说，我们支持的语法规则还比较有限，所需要的语义分析功能也有很多缺失。不过，词法分析仍然有一些功能不够充分，比如**数字字面量和字符串字面量等**。

在数字字面量这边，我们目前虽然已经支持比较简单的整数和浮点数的字面量。但还有二进制、八进制和十六进制的数字、科学计数法表达的数字，我们都还没去支持。而且，字

符串字面量上，我们也不支持 Unicode 和转义字符，并且字符串只能用双引号，还没使用单引号的版本。

接着，我们再来看**语法分析**功能。我们目前使用的都是一些比较高频的语法规则，**忽略了一些比较低频的语法**。比如，目前函数的参数只支持固定的参数数量，不支持变动数量的参数，也不支持参数的缺省值。再比如，我们目前的循环语句只支持 for 循环，并且不支持对集合的枚举，等等。

所以说，我们的词法分析和语法分析都还有不少功课需要去补呢。不过，我目前并不着急补这两方面。我的计划是，随着课程的推进，每当我们需要增加新特性的时候，就扩展一下这方面所需的词法和语法分析功能就好了。这样能够让你见证到像计算机语言这样的高难度软件一步步迭代成熟的过程，增强你自己驾驭类似的软件的信心。

**既然词法和语法分析功能都不是我们这节课的重点，那语义分析功能自然就是重点了。**

这是因为，实际上，在我们实现编译器的前端功能的时候，语义分析的工作量是最大的，但我们目前实现的功能确实有限。如果你有兴趣，可以参考我在《编译原理实战课》中对 Java 前端编译器的分析。在把编译工作分成的多个阶段中，大部分阶段都是去做语义分析相关的工作。

那我们现在的语义分析功能做到哪一步了呢？

在前面的课程中，我们已经实现了一些必要的语义分析功能，比如建立符号表、进行引用消解、分析哪个表达式是左值，以及进行简单的类型检查等等。不过这些功能其实还远远不够，因为还有很多潜在的语义错误没有被检查出来，因此需要我们逐步把这些工作补上。

在这个过程中，你会学习如何把数据流分析技术、类型计算技术用于语义分析工作。今天这节课，我们就先主要聚焦在数据流分析技术上。接下来，我们就举几个典型的场景，来学习如何在语义分析中使用数据流分析技术。

## 场景一：代码活跃性分析之程序是否 return 了？

我们在写函数的时候，如果这个函数需要返回值，那么在编译时，编译器会检查一下，是不是你所有的程序分支都以 return 语句结尾了。如果没有，编译器就会报错。我们举个例

子：

复制代码

```
function foo(a:number):number{
    if (a > 10){
        let b:number = a+5;
        return b;
        b = a + 10; //这段代码不可到达。
    }
}
```

你可以看一下，这段代码有什么问题呢？

首先，你会发现，这段代码里只有在 if 语句块有 return 语句。所以，当不满足 if 条件的时候，程序的执行流程就不会遇到这个 return 语句。那根据 TypeScript 的语义，此时的返回值是 undefined。而函数的返回值类型里呢，又不包含 undefined。所以这时，如果你用“tsc --strict example_return.ts”命令去编译它，tsc 会报下面的错误：

```
➜  22 git:(master) ✗ tsc example_return.ts --strict
example_return.ts:5:24 - error TS2366: Function lacks ending return statement
 and return type does not include 'undefined'.

5 function foo(a:number):number{
                         ~~~~~~
```

当然，如果函数的前面是下面的样子，在返回值里包含 undefined，那就是正确的。

复制代码

```
function foo(a:number):number|undefined
```

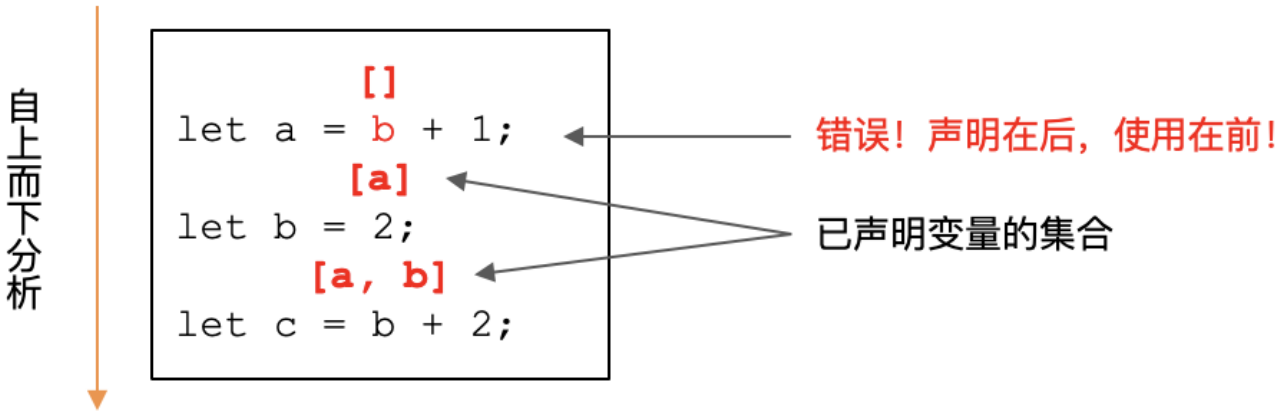
好，这是我们从示例代码中发现的第一个问题。那么第二个问题是什么呢？

你会看到，在 return 语句的下面还有一行代码“b = a + 10”，这一行代码其实是永远也不会被执行的。当然，这并不是一个错误，用 tsc 来编译也不会报错。但是，编译器或 IDE 工具最好要能够检查出这些问题，给程序员以提示。在编译生成代码的时候，编译器也可以直接把这些代码优化掉。

那如何检查出上面这些语义问题呢？那又需要用到**数据流分析技术**了。

到目前为止，我们已经多次接触到数据流分析技术了。在进行变量引用分析的时候，我们就曾实现过一个功能，检查出“变量是否在声明前就被引用”的错误。

它的处理逻辑是：语义分析程序遍历整个 AST，相当于自上而下地分析每一条代码。当程序遇到变量声明节点的时候，就会标记该变量已经被声明了。而当程序遇到变量引用节点时，如果它发现该变量虽然属于某个作用域，但它当前还没有被声明，那么它就会报语义分析错误。具体的实现，你可以参考 RefResolver 类中的代码。




另外，在实现寄存器分配算法时，我们也曾经使用过数据流分析技术，来计算每个变量的生存期，从而确定多个变量如何共享寄存器。在那个时候，我们是在 CFG 上进行数据流分析的，并且分析方向是自下而上的顺序。

针对我们前面实操过的这两个例子，你可以总结出来数据流分析的几个特点：

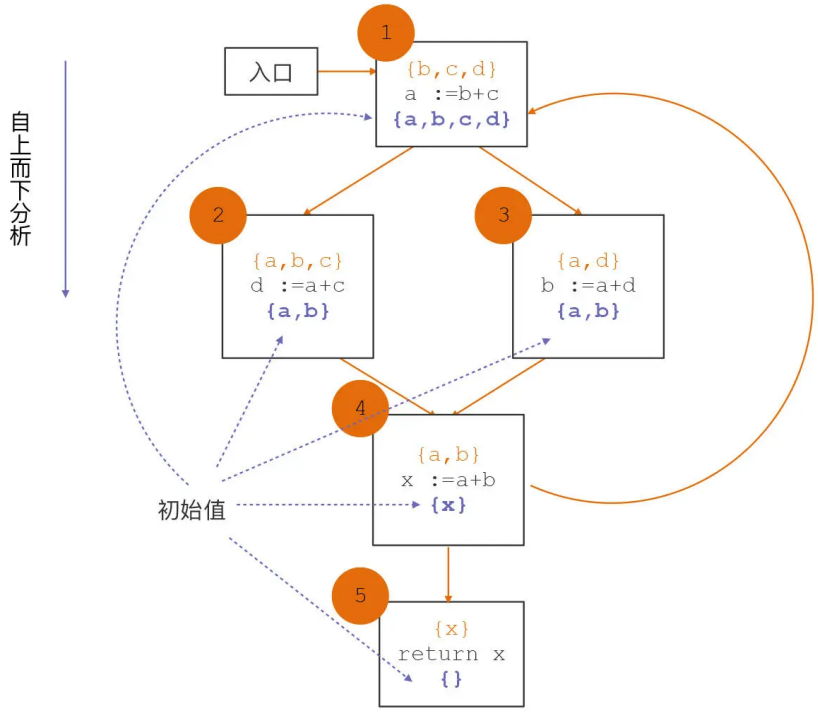
**首先，数据流分析技术可以用在像 AST 和 CFG 等多种数据结构上**，未来你还会见到我们把它用到其他的数据结构上；

**第二，针对不同的分析任务，数据流分析方向是不同的**，有的是自上而下，有的是自下而上，你需要确定清楚；

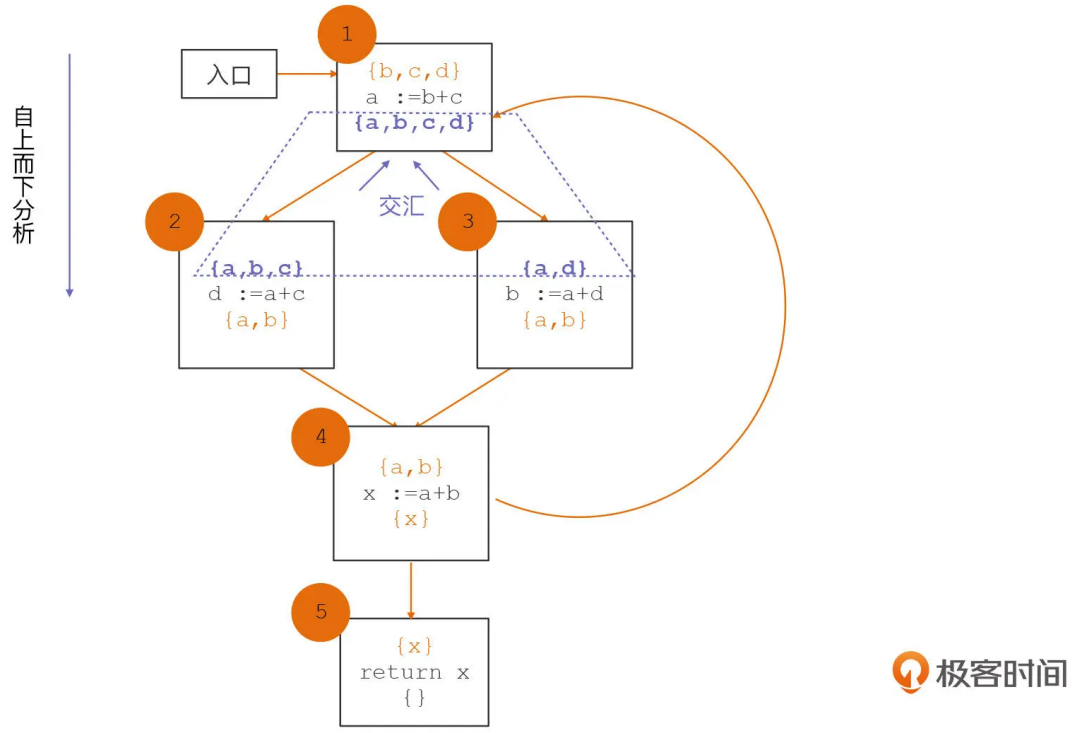
**第三，数据流分析的过程，都会针对一个分析变量，并会不断改变这个变量的值。**分析变量可能是一个单个的值，或者叫做标量，也可能是一组数值，比如向量和集合。在我们的前面的两个例子中，这个变量都是集合。第一个例子的分析变量是“已声明的变量的集合”，第二个例子的分析变量是“活跃变量集合”。

**第四，我们需要有一个规则或函数，基于这个规则来处理每行代码，从而计算新的变量值。**比如，在变量活跃性分析中，这个规则是只要遇到变量使用的语句，就往集合里添加该变量，遇到变量声明的语句，就从集合中去掉该变量。

**第五，要确定变量的初始值。**在第一个例子中，初始值是一个空集。在第二个例子中，每个基本块可能会有一个活跃变量的初始值，这些初始值是由 CFG 中的其他基本块决定的。



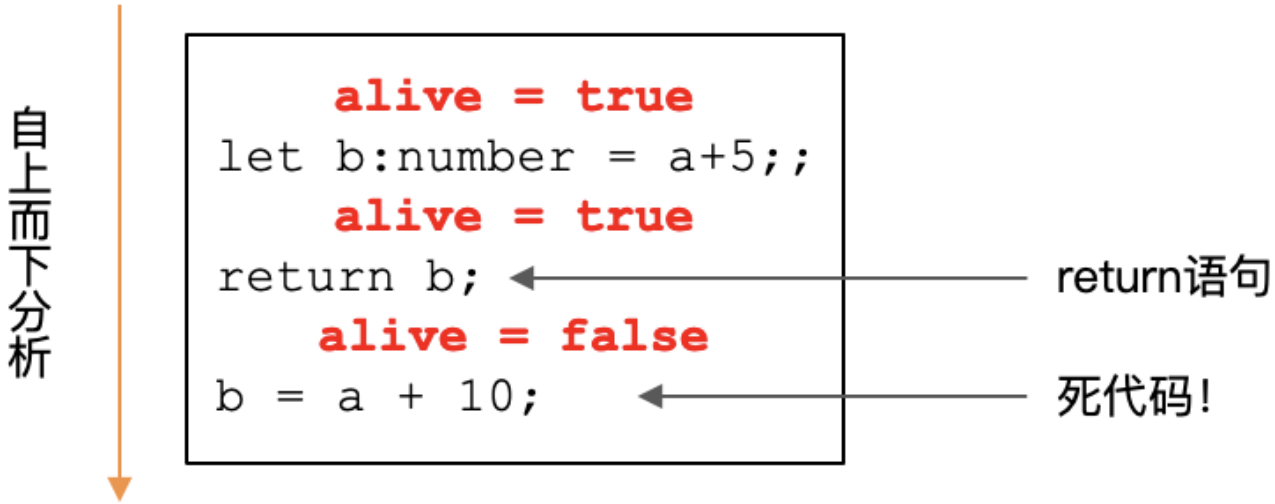

**还有最后一个共性，它们都有交汇函数。**交汇函数是用来在多个控制流交汇的时候，计算出交汇的值。在第二个例子中，当两个基本块交汇的时候，活跃变量集合是取两个集合的并集。




好了，上面这些就是数据流分析技术的核心特点。抓住这些核心特点，我们可以把这个技术用于更多的场景。比如说，我们就可以用这些特性解决上面这个程序是否正确 return 的问题。

在开始解决这个问题之前，我们先来梳理一个分析框架，看看我们具体要从哪些方面着手。

我们可以把一个程序在执行过程中是否遇到了 return 语句，用一个变量来描述，就是当前执行流程是不是 alive 的。我们从程序的开头，一行行代码的往下分析。在一开始，alive 的初始值是 true。当遇到 return 语句以后，alive 就变成了 false。



对于分支语句，比如 if 分支语句，则需要每个分支都要遇到一个 return 语句。如果一个分支的 alive 值是 alive1，另一个分支的 alive 值是 alive2，那么合起来的 alive 值是什么呢？是 alive1 || alive2。也就是说，必须每个分支都遇到 return 语句后，总的 alive 才是 false。这就是我们的交汇函数。

现在你应该就明白了，我们这次的数据流分析框架中需要具备这几个关键要素：

变量：alive，代表代码是否是活的，也就是是否遇到了 return。

初始值：true；

变换函数：当遇到 return 的时候，alive 变为 false;

交汇函数：逻辑或运算；

分析方向：自上向下。

具体怎么来解决这个问题呢？你可以参考 LiveAnalyzer的代码。我也把几段有代表性的代码放在了文稿里。

**第一段代码，是对语句块的分析。**在语句块中，编译器按自上向下的顺序检查每条语句。如果遇到 return，就把 alive 改成 false。对于 return 后面的语句，则作为“死代码”检测了出来。甚至，我们还可以修改 AST，直接把死代码从程序中去掉。这也是一种优化技术，名字就叫做“死代码消除”。

当然，死代码不仅仅指 return 之后的语句，有些 return 之前的语句可能也是死代码，是执行和不执行都没有影响的。如果你想了解这些，可以额外找些资料，也可以在留言区和我交流一下。

复制代码

```
visitBlock(block:Block):any{
    let alive:boolean = true;
    let deadCodes:Statement[] = []; //死代码
    for (let stmt of block.stmts){
        if (alive){
            alive = this.visit(stmt) as boolean;
        }
        //return语句之后的语句，都是死代码。
        else{
            //作为Warning，而不是错误。
            this.addWarning("Unreachable code detected.",stmt);
```

```
            deadCodes.push(stmt);
        }
    }

    //去除死代码
    for (let stmt of deadCodes){
        let index = block.stmts.indexOf(stmt);
        block.stmts.splice(index,1);
    }
    return alive;
}
```

**第二段代码，是对 If 分支语句的分析。**核心代码是 3 句，体现了多个分支的交汇逻辑。也就是说，只有两个分支都是 false 的情况下，总的计算结果才是 false。

复制代码

```
let alive1 = this.visit(ifStmt.stmt) as boolean;
let alive2 = ifStmt.elseStmt == null ? true : (this.visit(ifStmt.elseStmt) as
alive = alive1 || alive2;  //只有两个分支都是false，才返回false；
```

你可以用 node play example_return.ts 命令，编译一下试一试。我们的编译器就会报一个 Error 和一个 Warning，你可以看一下截屏。

```
➜  22 git:(master) ✗ node play example_return.ts
Warning: @(ln:9, col:9, pos:146) : Unreachable code detected.
Error: @(ln:5, col:1, pos:44) : Function lacks ending return statement and
return type does not include 'undefined'.

共发现0个语法错误，1个语义错误。
```

好了，现在我们已经解决了与 return 有关的分析工作了。那还有没有别的场景可以发挥数据流分析技术的威力呢？

有的。我们再来做一个变量赋值的分析看看。

## 场景二：变量赋值分析

什么是变量赋值的分析呢？我们看下面这个例子。

复制代码

```
function foo(a:number):number{
    let b:number;
    let c = a+b;
    if (a > 10){
        b = 1;
    }
    // else{
    //     b = 2;
    // }
    let d = a-b; //如果前面加上else部分，那么b就是被赋值的。
    return c+d;
}
```

在这个示例程序中，我们在第一个行声明了一个变量 b，但并没有给它赋值。接着，在第二个语句中，我们引用了 b。另外，在 if 语句之后，let d = a-b 语句也引用了变量 b。

你如果运行命令 tsc --strict example_assign.ts，那么 tsc 编译器会报错： “Variable ‘b’ is used before being assigned.”。

```
➜  22 git:(master) ✗ tsc --strict example_assign.ts
example_assign.ts:6:15 - error TS2454: Variable 'b' is used before being as
signed.

6     let c = a+b;
              ~

example_assign.ts:10:15 - error TS2454: Variable 'b' is used before being a
ssigned.

10     let d = a-b;
               ~


Found 2 errors.
```

相信你看完这个例子就明白了，变量赋值分析要解决的就是变量被赋值之前就已经被使用的问题。这个问题也适合用数据流分析技术来解决，因为我们只要顺序查找对变量进行赋值和使用的代码，就能知道是否存在错误了。

**那么，我们该如何套用前面说过的数据流分析框架，来解决赋值分析问题呢？**

---

首先是**分析变量**。这里分析变量就是在每一条语句那里，每个变量是不是肯定被赋值了。我们可以用一个 Map 来记录每个变量的赋值状态。

复制代码

```
assigned:Map<VariableSym,boolean> = new Map();
```

第二，**初始值**。在一开始，每个变量都没有被赋值。

第三，**变换函数**。我们每次遇到给一个变量赋值语句或者变量初始化语句，都就可以改变该变量的赋值状态。

第四，**交汇函数**。在遇到分支的时候，我们必须保证每个分支都给这个变量赋值了，那么这个变量才肯定赋值了。

第五，**分析方向**。程序自上向下进行分析。

具体实现你可以参见AssignAnalyzer。这个程序比前一个要稍微复杂一点，我在这里也给你标出了一些核心步骤。

首先，我们要一次记录多个变量是否被赋值的信息，我这里用了一个 map。

复制代码

```
//每个变量的赋值情况
assignMode:Map<VarSymbol, boolean> = new Map();
```

接着，要在变量声明的时候，我们把变量加到这个 map 里。如果变量在声明时就被初始化了，那么记录它的赋值状态为 true。如果通过单独的赋值语句被赋值，赋值状态也会被改成 true。

复制代码

```
//变量声明中可能会初始化变量
visitVariableDecl(variableDecl: VariableDecl):any{
    if (variableDecl.init != null) this.visit(variableDecl.init);
    //如果有初始化部分，那么assigned就设置为true
    this.assignMode.set(variableDecl.sym as VarSymbol, variableDecl.init != nu
}

//处理赋值语句
visitBinary(binary:Binary):any{
    if (Operators.isAssignOp(binary.op)){
```

```
        this.visit(binary.exp2); //表达式右侧要照常遍历，但左侧就没有必要了。
        if (typeof (binary.exp1 as Variable).sym == 'object'){
            let varSym = (binary.exp1 as Variable).sym as VarSymbol;
            this.assignMode.set(varSym, true);
        }
    }
    else{
        super.visitBinary(binary);
    }
}
```

接着，你要在变量被引用的地方，基于该 map 检查该变量是否被初始化了。

复制代码

```
//变量声明中可能会初始化变量
visitVariable(variable: Variable):any{
    let varSym = variable.sym as VarSymbol;
    if (this.assignMode.has(varSym)){
        let assigned = this.assignMode.get(varSym) as boolean;
        if (!assigned){
            this.addError("variable '" + variable.name + "' is used before bei
        }
    }
    else{
        console.log("whoops,不可能到这里@semantic.ts/visitVariable");
    }
}
```

再然后，我们仍然要处理流程分枝的情况。这个时候，要针对每个分枝计算一个 assignMode 对象，然后合并每个分枝的结果。

复制代码

```
visitIfStatement(ifStmt:IfStatement):any{
    //算法：把assignMode克隆两份，分别代表遍历左支和右支的结果，然后做交汇运算
    let oldMode = this.cloneMap(this.assignMode);
    //遍历if块
    this.visit(ifStmt.stmt);
    let mode1 = this.assignMode;
    //遍历else块
    this.assignMode = this.cloneMap(oldMode);
    if (ifStmt.elseStmt != null) this.visit(ifStmt.elseStmt);
    let mode2 = this.assignMode;
    //交汇运算
    this.assignMode = this.merge(mode1, mode2);
}
```

最后一个细节，是要忽略死代码。如果变量是在死代码里被赋值的，那仍然不算数，因为实际的控制流并不会到达这里。

复制代码

```
//略过死代码
visitBlock(block:Block):any{
    for (let stmt of block.stmts){
        let alive = this.visit(stmt) as boolean;
        if(typeof alive == 'boolean'){
            //如果遇到return语句，后面的就是死代码了，必须忽略掉。
            return;
        }
    }
}

//检测return语句
visitReturnStatement(rtnStmt: ReturnStatement):any{
    if(rtnStmt.exp != null)  this.visit(rtnStmt.exp);
    return false;  //表示代码活跃性为false
}
```

## 课程小结

好了，今天的内容就是这些。我带你回顾一下本节的重点：

首先，我们回顾了一下 PlayScript 当前的编译器前端功能的现状。词法分析和语法分析这两部分的功能呢，目前基本够用。如果增加新的语言特性，那我们就迭代增加就好了。而语义分析功能呢，我们却必须要做一些额外的补充，来检查出程序中的一些错误。

第二，在补充语义分析功能的过程中，我们这节课重点使用了数据流分析方法。我们之前就提过数据流分析方法很有用，相信你在这节课里也体会到了。每次采用数据流分析方法的时候，你都要识别出五个关键的要素：变量、变量的初始值、变换函数、交汇函数和分析方向。只要你识别出这五个要素，那么编程实现就比较简单了。

第三，我们今天进行了代码活跃性分析和赋值分析。代码活跃性分析能够分析出程序的控制流是否都匹配了 return 语句，还能分析出一些死代码，并进行优化。赋值分析能够分析出变量是否在赋值之前就被使用的错误。

下一节课，我们还会介绍另一个重要的技术，就是**类型计算**，这会进一步增强我们 PlayScript 的语义分析功能。

## 思考题

我们今天提到，语义分析的功能是有很多的。你能否说一说，到目前为止，我们还欠缺哪些语义分析功能呢？欢迎在留言区和我分享。

欢迎你把这门课分享给更多对语义分析感兴趣的朋友。我是宫文学，我们下节课见。

## 资源链接

这节课的示例代码在这里！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | 加深对栈的理解：实现尾递归和尾调用优化

下一篇 23 | 增强编译器前端第2步：增强类型体系

## 精选留言 (2)

写留言

**静心**

2021-09-27

除了return，抛出异常也是一种常见的情况，对于一门儿高级语言，异常的处理对于编译工具来说应该也是一种不小的挑战吧？
再有就是重复定义或声明变量，也是一种情况。

展开

**奋斗的蜗牛**

2021-09-27

佩服，这些知识经过老师的讲解，竟然如此有趣